Chapter 8

Functions and Files

In this Chapter, we tie up some loose ends concerning some of the built in functions provided by the C language. In previous chapters we have been using such functions in our programming examples to do data input and output; functions such as `scanf()`, `printf`, `getchar()`, and `putchar()`. These routines are part of a library of standard routines. As we have seen, we can use these functions by including the header file in which they are declared (in this case <`stdio.h`>). These header files contain the prototypes for functions as well as macros that are needed for their use.

Previously, when we have needed routines for other operations (e.g. testing if a character is a digit), we have written our own. Such operations are common enough in C programs that the implementors have included predefined routines to perform them. These routines are collectively called the C Standard Library. We begin this Chapter by describing a few other built in functions provided in the Standard Library, describing their use and using them in a few sample programs. A longer (though not complete) listing of the Standard Library, together with descriptions, is provided in Appendix C.

We next give a more thorough description of our I/O functions, `scanf()` and `printf()`. Finally, we discuss variations of the standard I/O routines, which allow direct access to data stored in files.

8.1 The C Standard Library

We have already used several I/O routines from the standard library: scanf(), printf(), getchar() and putchar(). Many other useful routines are provided in one or more libraries supplied with the compiler or in header files. When a function in one of these libraries is used, the name of the library must be supplied to the linker. Otherwise, the linker is unable to resolve the reference to that function. If the function resides in the standard library, the linker does not need to be supplied the name. The linker always searches the standard library by default for any unresolved functions used in the program.

Standard header files supplied with the compiler declare function prototypes for standard library functions in several categories. They also define data types, symbolic constants, and macros. Header files must be included in the source program if any of the definitions, macros, or function prototypes declared in them are to be used.

Many of the functions we have defined in our example programs are available either as standard macros in a header file or as functions in the standard library. We could have used these standard

routines in many of our examples. However, we wrote our own versions because it is instructive to see how functions are written.

The following are descriptions of some of the commonly used routines. Similar descriptions of other routines will be provided as we use them. A listing of ANSI standard library routines is provided in Appendix C. It must be understood that the standard is a suggested standard, and all vendors of C compilers may not follow the suggested standard exactly.

The listing below specifies what header file must be included, if any, before the routine listed may be used. It also specifies which file contains the prototypes, if applicable.

8.1.1 Character Processing Routines

Character Classification Routines

is... *Prototype:*

```
int isalnum(int c);
int isalpha(int c);
int isascii(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
```

in: <ctype.h>

Returns:

isalnum	TRUE if c is a letter or a digit. ('A'-'Z', 'a'-'z', '0'-'9').
isalpha	TRUE if c is a letter. ('A'-'Z', 'a'-'z')
isascii	TRUE if c is in the range 0-127.
iscntrl	TRUE if c is a control character. (0-31, 127)
isdigit	TRUE if c is a digit. ('0'-'9')
isgraph	TRUE if c is a graphical character, i.e. a printable character except for space. (33-126)
islower	TRUE if c is a lower case letter. ('a'-'z')
isprint	TRUE if c is a printable character. (32-126)
ispunct	TRUE if c is a punctuation character.
isspace	TRUE if c is a space, tab, newline, or any white space character. (9-13, 32)
isupper	TRUE if c is an upper case letter. ('A'-'Z')
isxdigit	TRUE if c is a hexadecimal digit. ('0'-'9', 'A'-'F', 'a'-'f')

Description: These are macros that classify a character, c, given as an integer type (ASCII) value. They return non-zero if TRUE and zero if FALSE.

Character Conversion Routines

toascii *Prototype:* `int toascii(int c);` *in:* <ctype.h>

Returns: converted value of `c`.

Description: Converts an integer, `c`, to ASCII format by clearing all but the lower seven bits. The value returned is in the range 0-127.

tolower *Prototype:* `int tolower(int c);` *in:* < ctype.h>

Returns: Lower case value of `c` if `c` was upper case, `c` otherwise.

toupper *Prototype:* `int toupper(int c);` *in:* < ctype.h>

Returns: Converts `c` to upper case if `c` is lower case; otherwise, `c` is left unchanged.

Note that all the above library character routines use an `int` type argument. Since the value of a character is its ASCII value of type `int`, passing a `char` type argument to these routines is the same as passing an `int` type ASCII value.

Character Routines Programming Examples

Let us use some of the above library routines to write a variation on our previous program to pick out words in the input text. The revised program only picks out valid words; namely identifiers. We will assume that a valid identifier starts with a letter and may be followed by any number of letters and/or digits. White space delimits an identifier; otherwise, it is ignored. Any character that does not belong in an identifier is an illegal character; and also delimits an identifier.

We will need to test each character to see if it is a letter, a digit, a white space, etc. We will use library functions `isalpha()`, `isalnum()`, and `isspace()` to test for these characters. The descriptions for them states that we must include file <ctype.h>. In addition to finding and printing identifiers, the program also keeps a count of them.

The only change in the previous algorithm is that now we start a word if and only if it starts with a letter. Once a word is started, it continues as long as characters are letters or digits; otherwise, the word is terminated and counted. The program is shown in Figure 8.1.

We test if the first character after white space is a letter. If so, we build an identifier. Otherwise, if it is `EOF`, we terminate the loop. Otherwise, it must be an illegal character.

Sample Session:

```
***Print Identifiers***

Type text, terminate with EOF
Programming is easy
Programming
is
easy
once an algorithm is developed
once
an
algorithm
```

```
/*   File: ident.c
     Program reads characters one at a time until EOF. It prints out
     each identifier in the input text and counts the total number of
     identifiers. It ignores white space except as a delimiter for an
     identifier. An identifier starts with an alphabetic letter and may be
     followed by any number of letters or digits. All other characters are
     considered illegal.
*/

#include <stdio.h>
#include <ctype.h>

main()
{    int cnt = 0;
     signed char c;

     printf("***Print Identifiers***\n\n");
     printf("Type text, terminate with EOF ^Z or ^D)\n");
     c = getchar();
     while (c != EOF) {
          while (isspace(c))            /* skip leading white space   */
               c = getchar();

          if (isalpha(c)) {         /* if a word starts with a letter */
               while (isalnum(c)) {     /* while c is letter or digit */
                    putchar(c);         /* print c                    */
                    c = getchar();      /* read next char             */
               }
               putchar('\n');        /* end identifier with a newline */
               cnt++;                   /* increment cnt              */
          }

          else if (c == EOF)            /* if end of file             */
               break;                   /* break out of loop          */

          else {                  /* otherwise, it is an illegal char */
               printf("Illegal character %c\n", c);
               c = getchar();
          }
     }
     printf("Number of Identifiers = %d\n", cnt);
}
```

Figure 8.1: Code for `ident.c`

```
is
developed
^D
Number of Identifiers = 8
```

8.1.2 Math Routines

There are many mathematical routines in the standard library, such as `abs()`, `pow()`, `sqrt()`, `rand()`, `sin()`, `cos()`, `tan()`, and so forth. The prototypes for these are defined in the header file, <`math.h`>, which must be included whenever these functions are used in a program. In addition, on Unix systems, the math library is maintained separately from the standard library, thus requiring that it be linked when the code is compiled. This can be done with the compiler command:

```
cc  filename.c -lm
```

The option `-l` specifies that a library must be linked with the code and the `m` specifies the math library. Note that this option *MUST* appear as the last item on the command line.

Most of the functions listed above are self explanatory (and are described in detail in Appendix C). As an example, let us look at the function, `rand()` which generates pseudo-random integers in the range of numbers from 0 to the largest positive integer value. The numbers cannot be completely random because the range is limited. However, for the most part, the numbers generated by `rand()` appear to be quite random. The prototype for the function is:

```
int rand(void);
```

Each time the function is called, it returns a random integer number. Figure 8.2 shows an example which generates and prints some random numbers.

Sample Session:

```
346
130
10982
1090
11656
```

The random number generator will always start with the same number unless it is "seeded" first by calling the function, `srand()`. The prototype for it is:

```
void srand(unsigned x);
```

In the example in Figure 8.3, we seed the random number generator with a user supplied number. The program then finds random throws for a single dice. After the random generator is seeded, every random number generated, `n`, is evaluated modulo 6, i.e. `n % 6` is evaluated. This results in numbers from 0 to 5. We add one to obtain the dice throws from 1 to 6.

Sample Session:

```
***Single Dice Throw Program***
```

```
/* File: rand.c
     Program uses random number generator to print some random
     numbers.
*/
#include <stdio.h>
#include <math.h>
main()
{    int i;
     int x;

     for (i = 0; i < 5; i++) {
          x = rand();
          printf("%d\n", x);
     }
}
```

Figure 8.2: Small program to generate random numbers

```
/* File: dice.c
     Program throws a single dice repeatedly.
*/
#include <stdio.h>
#include <math.h>

main()
{    int i, d1;

     printf("***Single Dice Throw Program***\n\n");
     printf("Type a random unsigned integer to start: ");
     scanf("%d", &i);
     srand(i);

     for (i = 0; i < 5; i++) {
          d1 = rand() % 6 + 1;
          printf("throw = %d\n", d1);
     }
}
```

Figure 8.3: Program for generating random dice values

```
Type a random unsigned integer to start:  12737
throw = 2
throw = 6
throw = 1
throw = 3
throw = 5
```

Similarly, we can write a program that draws a card from a full deck of 52 cards as shown in Figure 8.4. It starts by seeding the random number generator before its use. Next, a random number is generated and evaluated modulo 52, resulting in a random number between 0 and 51, representing a card. For a number, `n`, the value `n / 13` is in the range 0 through 3, each corresponding to a suit: say 0 is clubs, 1 is diamonds, 2 is hearts, and 3 is spades. In addition, `n % 13 + 1` evaluates to a number in the range 1 through 13, corresponding to a card in a suit: say 1 is ace, 2 is deuce, ..., 11 is jack, 12 is queen, and 13 is king.

Sample Session:

```
***Single Card Draw Program***

Type a random unsigned integer to start:  30257
Diamond 2
Heart Jack
Heart 3
Diamond Queen
Diamond 10
```

The next program uses the library function, `sqrt()`, to obtain square roots of randomly generated numbers. The function, `sqrt()`, requires its argument to be of type `double`, and it returns type `double`. In the program shown in Figure 8.5, the randomly generated whole number is assigned to a double variable before finding its square root.

Sample Session:

```
***Square Root Program - Random Numbers***

Sq.Rt.  of 346.000000 is 18.601075
Sq.Rt.  of 130.000000 is 11.401754
Sq.Rt.  of 10982.000000 is 104.795038
Sq.Rt.  of 1090.000000 is 33.015148
Sq.Rt.  of 11656.000000 is 107.962957
```

These have been just a few examples of using routines available in the math library. A complete listing of math routines is provided in Appendix C. Rather than writing our own functions all the time, we will make use of library functions in our code wherever we can in the future.

8.2 Formatted Input/Output

We have been using the I/O built-in functions `printf()` and `scanf()` which are the primary routines for formatted output and input in C (the "f" stands for formatted). We have already

```
/* File: card.c
     Program draws a card each time from a full deck of 52 cards.
*/
#include <stdio.h>
#include <math.h>

#define CLUB 0
#define DIAMOND 1
#define HEART 2
#define SPADE 3

#define ACE 1
#define JACK 11
#define QUEEN 12
#define KING 13

main()
{    int i, d1, card, suit;

     printf("***Single Card Draw Program***\n\n");
     printf("Type a random unsigned integer to start: ");
     scanf("%d", &i);
     srand(i);                /* seed the random number generator */
     for (i = 0; i < 5; i++) {
          d1 = rand() % 52;        /* draw a card                */
          suit = d1 / 13;          /* find the suit 0,1,2,3      */
          card = d1 % 13 + 1;      /* find the card 1, 2, ..., 13 */

          switch (suit) {          /* print suit                 */
               case CLUB: printf("Club "); break;
               case DIAMOND:  printf("Diamond "); break;
               case HEART: printf("Heart "); break;
               case SPADE: printf("Spade "); break;
          }
          switch (card) {         /* print the card within a suit */
               case ACE: printf("Ace"); break;
               case JACK: printf("Jack"); break;
               case QUEEN: printf("Queen"); break;
               case KING: printf("King"); break;
               default: printf("%d", card);
          }
          printf("\n");
     }
}
```

Figure 8.4: Program for randomly picking a card

```
/*   File sqrt3.c
     Program computes and prints square roots of numbers randomly
     generated.
*/
#include <stdio.h>
#include <math.h>

main()
{    int i;
     double x;

     printf("***Square Root Program - Random Numbers***\n\n");
     for (i = 0; i < 5; i++) {
          x = rand();
          printf("Sq.Rt. of %f is %f\n", x, sqrt(x));
     }
}
```

Figure 8.5: Code for finding the square root of random numbers

discussed many of the conversion specifications; we now present a more complete description of the formatted I/O functions together with examples. (While our discussion here concerns `printf()` and `scanf()`, it applies equally well to conversion specifications for `fprintf()` and `fscanf()` described in the next Section.

8.2.1 Formatted Output: `printf()`

As we have seen, `printf()` expects arguments giving a format string and values to be printed. The `printf()` prototype, in `stdio.h`, is:

```
int printf(char *, ...);
```

The first argument of printf() is the format string (we will see what the above type declaration means in the next chapter). The number of remaining arguments depends on the number of conversion specifiers in the format string. In C, an ellipsis, i.e. "...", is used to indicate an arbitrary number of arguments. The return value of `printf()` is an `int` giving the number of bytes output, if successful; otherwise it returns `EOF`. This information from `printf()` is not generally very useful, and we often simply ignore the return value.

The function, `printf()`, converts, formats, and prints its arguments on the standard output using the conversion specifications given in the format string. The format string is made up of two kinds of characters: regular characters, which are simply copied to the output, and conversion specification characters. A conversion specification indicates how the corresponding argument value is to be converted and formatted before being printed. The number of conversion specifications in the format string must match exactly the number of arguments that follow; otherwise, the results are undefined. The data type of the argument should also match the data type it

will be converted to; for example, integral types for decimal integer formats, `float` or `double` types for floating point or exponential formats, and so on. If the proper type is not used, the conversion is performed anyway *assuming* correct data types and the results can be very strange and unexpected. Of course, character values are integral types; so characters can be converted to ASCII integer values for printing, or printed as characters. We have already seen most of the conversion characters. Table 8.1 gives a complete list with their meanings. We will discuss some examples, given the following declarations and initializations:

```
int i;
char c;
float f1;
double d1;
char *s;
long x;

i = 33;
c = 'e';
f1 = 12345.00
d1 = 12345.00
s = "This is a test";
x = 123456789;
```

Different conversion characters may be used to print the values of these variables. The space used to print a value is called the **field**, and by default, is exactly the space needed to print the value. We show examples of conversion characters and default output below:

Conversion Specifier	Variable	Output
`%d`	`i`	*33*
`%o`	`i`	*41*
`%x`	`i`	*21*
`%u`	`i`	*33*
`%c`	`c`	*e*
`%d`	`c`	*101*
`%c`	`i`	*!*
`%s`	`s`	*this is a test*
`%f`	`f1` or `d1`	*12345.000000*
`%e`	`f1` or `d1`	*1.234500E+004*
`%g`	`f1` or `d1`	*12345*

So far, we have used very simple conversion specifiers, such as `%d`, `%f`, and `%c`. A complete conversion specification starts with the character `%` and ends with a conversion character. Between these two characters, special format characters may be used which can specify justification, field width, field separation, precision, and length modification. The characters that follow the `%` character and precede the conversion characters are called **format characters**. All format characters are optional, and if they are absent their default values are assumed. (We will indicate the default value in each case below). The syntax of a complete conversion specifier is:

Character	Conversion
`d`	The argument is taken to be an integer and converted to decimal integer notation.
`o`	The argument is taken to be an integer and converted to unsigned octal notation without a leading zero.
`x`	The argument is taken to be an integer and converted to unsigned hexadecimal notation without a leading `0x`.
`u`	The argument is taken to be an unsigned integer and converted to unsigned decimal notation.
`c`	The argument is taken to be an ASCII character value and converted to a character.
`s`	The argument is taken to be a string pointer. Unless a precision is specified as discussed below, characters from the string are printed out until a `NULL` character is reached. (Strings will be discussed further in the next chapter).
`f`	The argument is taken to be a `float` or `double`. It is converted to decimal notation of the form `[-]ddd.dddddd`, where the minus sign shown in square brackets may or may not be present. The number of digits, `d`, after the decimal point is 6 by default if no precision is specified. The number of digits, `d`, before the decimal is as required for the number.
`e`	The argument is taken to be a `float` or `double`. It is converted to decimal notation of the form `[-]d.ddddddE[+/-]xxx`, where the leading minus sign may be absent. There is one digit before the decimal point. The number of digits, `d`, after the decimal point is 6 if no precision is specified. The `E` signifies the exponent, ten, followed by a plus or minus sign, followed by the exponent. The number of digits in the exponent, `x`, is implementation dependent, but not less than two.
`g`	The same as `%e` or `%f` whichever is shorter and excludes trailing zeros.

Table 8.1: Conversion Specifier Characters for `printf()`

%-DD.ddlX

where X is one of the conversion characters from Table 8.1. The other format characters must appear in the order specified above and represent the following formatting information: (the corresponding characters are shown in parentheses).

Justification (-) The first format character is the minus sign. If present, it specifies left justification of the converted argument in its field. The default is right justification, i.e. padding on the left with blanks if the field specified is wider than the converted argument.

Field Width (DD) The field width is the amount of space, in character positions, used to print the data item. The digits, DD, specify the *minimum* field width. A converted argument will be printed in a field of at least this size, if it fits into it; otherwise, the field width is made large enough to fit the value. If a converted argument has fewer characters than the field width, by default it will be padded with blanks to the left, unless left justification is specified, in which case, padding is to the right.

Separator (.) A period is used as a separator between the field width and the precision specification.

Precision (dd) The digits, dd, specify the precision of the argument. If the argument is a `float` or `double`, this specifies the number of digits to the right of the decimal point. If an argument is a string, it specifies the maximum number of characters to be printed from the string.

Length Modifier (l) The length modifier, l, (ell) indicates that an integer type argument is a `long` rather than an `int` type.

Some examples of format specifications using the previous variable types and values are shown below, where the field width is shown between the markers, | and |.

```
            Conversion    Variable   Output

 Field Pos.                          01234567890123456789
-----------------------------------------------------------
               %10d           i      |        33|
              %-10d           i      |33        |
                                     |          |
               %10f          f1      |12345.000000
              %-10f          f1      |12345.000000
              %20.2f         f1      |             12345.00|
             %-20.2f         f1      |12345.00             |
               %5c            c      |        E|
               %-5c           c      |E        |

               %10s           s      |This is a test
```

```
  %20s          s      |            This is a test|
 %-20s          s      |This is a test            |
%20.10s         s      |               This is a  |

  %ld           x      |123456789
 %-12ld         x      |123456789     |
```

8.2.2 Formatted Input: `scanf()`

Like `printf()`, `scanf()` expects its first argument to be a format string, but unlike `printf()`, the remaining arguments are *addresses* of the variables in which to put the data that is read. The prototype for `scanf()`, also in `stdio.h`, is:

```
int scanf(char *, ...);
```

As we've said, the returned value is the number of items read, or `EOF`. The format string controls the input order, conversion of the input data to the specified type, and format specification. Each conversion specification appearing in the format string is applied, in turn, to the next input data item in the input stream. After the specified conversion, the item is placed where the next succeeding argument points; so each of the arguments must be an address.

Besides the conversion specifications that start with `%`, the format string may also include regular characters. Regular *white space* characters in the format string are ignored. Any regular non-white space characters must be matched exactly in the input stream. For example:

```
scanf("x= %d", &x);
```

The input stream must include the characters `x=`, which are matched by the corresponding character in the format string, before an integer value is read. A valid sample input for this format string is:

x= 1234

The characters, *x=*, are first matched, then the integer, 1234, is read and assigned to the variable, `x`. If the characters, *x=*, are not matched in the input stream, no input is possible, and `scanf()` will return the value 0.

As before, a conversion specification starts with a `%` and ends with one of the conversion characters given in Table 8.2. Between `%` and the conversion character, there can be an optional assignment suppression character, `*`, followed by an optional number indicating the maximum field width. The maximum field width specifies that no more than that number of characters in the input stream may be used for the next data item. The converted result is stored where the corresponding argument points unless the assignment suppression character is used. If the suppression character is used, the result is discarded. The conversion characters with their meanings are given in Table 8.2. All of these except `c` and `s` may be preceded by the length modifier, `l` (ell), where, in the case of integral type data, the corresponding argument should be `long` and in the case of floating point data, the argument should be `double`. For example, with the following declarations:

```
int i, k;
char c;
```

Character	Conversion
`d`	The input is expected to be a decimal integer. The corresponding argument should be an integer address.
`o`	The input is expected to be an octal number. The corresponding argument should be an integer address.
`x`	The input is expected to be a hexadecimal number. The corresponding argument should be an integer address.
`c`	The input is expected to be a character. Any character including white space may be input without being skipped over. The corresponding argument should be a character address.
`s`	The input is expected to be a string of characters, and the corresponding argument should be a character pointer to an array of characters large enough to accommodate the string plus the terminating `NULL` character. (Arrays are discussed in Chapter 7). The input *will* skip over initial white space and will terminate when a white space character is encountered in the input stream.
`f`	The input is expected to be a floating point number and the corresponding argument should be a `float` address. The input may have a sign, followed by a string of digits, optionally followed by a decimal point and a string of digits, which may be followed by an E or e and a signed or unsigned integer exponent.
`e`	Same as f.

Table 8.2: Conversion Specifier Characters for `scanf()`

```
float f1;
double d1;
char s[80];
long x;
```

Consider the following statements with the input as shown below each statement.

```
scanf("Integer: %4d %f", &i, &f1);
```

Input is:

Integer: 1234567

First, the regular characters, *Integer:* are matched. Then a field of 4 is used to read the integer, *1234*. Finally, a `float`, *567.0* is read. The integer, 1234, will be stored in `i`, and 567.0 will be stored in `f1`.

```
scanf("%4s %*c %c", s, &c);
```

Surprises are everywhere

A field of 4 is used to read a string, `"Surp"`, which is placed in `s`. The next character, `'r'`, will be read and discarded, and the next character, `'i'`, will be stored in `c`.

```
scanf("%s %*s %d", s, &i);
```

Surprise number 1

This time the string `"Surprise"` will be stored in `s`, the next string `"number"` will be discarded, and the integer 1 will be stored in `i`.

8.3 Direct I/O with Files

So far all our programs have used standard files for input and output; normally the keyboard and screen. Unless the standard files are redirected, users must enter data as needed, which may become inconvenient or impractical as the amount of data gets large. However, if redirection is used to read input data from other files, then *ALL* input must come from redirected files; which means the programs cannot interact with the user. Practical programs require the ability to use files for I/O as well as to interact with users via standard files. For example, data may be needed repeatedly, by different programs, over a period of time. Such data should be stored in files on disks or other peripheral devices, and programs should be able to retrieve data from these files as needed. In addition, programs can save useful data into files for later use.

In this Section, we describe some variations on our previous Input/Output routines which behave similarly, but access data directly from or to files.

8.3.1 Character I/O

We have written programs for processing characters using the routines `getchar()` and `putchar()` which read or write single characters from or to the standard input or output. The standard library provides additional, more general, routines which read or write single characters from or to any file (including `stdin` or `stdout`). We will illustrate the use of these routines with two short examples.

Our next task is to read text input from a non-standard input file and compute the frequency of occurrence of each digit in the text:

FREQ: Read input from a specified text file and calculate the frequency of occurrence of each digit in the file.

Our task calls for us to read textual data from an input file. In order for the program to be able to read from a file, the file must be identified to the program. This process is called *opening* the file. Likewise, when our use of the data in a file is complete, the file should be *closed*. Opening a file informs the program where data is to be read from, and initializes a system data structure which keeps track of how far reading has progressed in the file (along with other information needed by the operating system). Most files in C programs are treated as a *stream* of characters by the library routines that access them, and so, an open file is sometimes also referred to as a stream. Closing a file relinquishes all use of the file from the program back to the operating system. When a file is opened, the input starts at the beginning of the file and continues until the end of file is reached. The standard files, `stdin` and `stdout`, behave the same way, but they are opened automatically at the beginning of the program. They cannot be re-opened and should not normally be closed.

We can now write an algorithm for our task of counting frequency of occurrence of digits in a file (or stream). We will use an array, `digit_freq[]` to store the frequency of each digit. For each character, `ch` read, if `ch` is a digit symbol, then `ch - '0'` is the numeric equivalent of that digit and we will use `digit_freq[ch - '0']` to store the frequency of the digit. That is, `digit_freq[0]` will store frequency of digit character `'0'`, `digit_freq[1]` will store frequency for `'1'`, and so on. Here is the algorithm:

```
initialize array digit_freq[] to zero
open input file
while NOT EOF, read a character from input file stream
     if a character ch is a digit
          increment digit_freq[ch - '0']
print results to standard output
close input file
```

We begin by initializing the array, `digit_freq[]` to zero and each time a digit character is encountered, an appropriate frequency is incremented. The program implementation is shown in Figure 8.6 and assumes that the file to be read is named `test.doc`.

The input file, `test.doc` consists of a single line shown below:

```
245  87  129  45  28
```

Sample Session:

```
***Digit Occurrence Counter***
```

```
/* File: cntdigits.c
    This program reads characters from a file stream and counts the
    number of occurrences of each digit.
*/
#define MAX 10
#include <stdio.h>
#include <ctype.h>      /* for isdigit() */
main()
{
    int digit_freq[MAX],i;
    signed char ch;
    FILE * fin;

    printf("***Digit Occurrence Counter***\n\n");
    /* initialize the array */
    for (i = 0; i < MAX; i++)
         digit_freq[i] = 0;

    fin = fopen("test.doc", "r");      /* open input file */
    if (!fin) {                        /* if fin is a NULL pointer */
         printf("Unable to open input file: test.doc\n");
         exit(0);                      /* exit program */
    }

    while ((ch = getc(fin)) != EOF) {  /* read a character into ch */
         if (isdigit(ch))              /* if ch is a digit */
              digit_freq[ch - '0']++;  /* increment count for digit ch */
    }
    fclose(fin);

    /* summarize */
    for (i = 0; i < MAX; i++)
         printf("There are %d occurrences of %d in the input\n",
                   digit_freq[i],i);
}
```

Figure 8.6: Code for Counting Digits

```
There are 0 occurrences of 0 in the input
There are 1 occurrences of 1 in the input
There are 3 occurrences of 2 in the input
There are 0 occurrences of 3 in the input
There are 2 occurrences of 4 in the input
There are 2 occurrences of 5 in the input
There are 0 occurrences of 6 in the input
There are 1 occurrences of 7 in the input
There are 2 occurrences of 8 in the input
There are 1 occurrences of 9 in the input
```

Let us first give a summary explanation. In the declaration section of the function, `main`, a *file pointer* variable, `fin`, is declared to be of type `FILE *`. The type `FILE` is defined using a `typedef` in <`stdio.h`> as a special data structure containing the information about a file need to access it. After the array, `digit_freq[]`, is initialized to zero, the file, `test.doc`, is opened using the standard library function, `fopen()`:

```
fin = fopen("test.doc", "r");
```

The function, `fopen()`, takes two arguments: a string which gives the name of the physical file, and a second string which specifies the mode (`"r"` (for read) indicates an input file). If the file can be opened, `fopen()` returns a file pointer which can be used to access the corresponding stream. If the file cannot be opened, `fopen()` returns a `NULL` pointer, so the program tests if the returned value of the file pointer, `fin`, is `NULL` and, if so, terminates the program after a message is printed. If the file opened (i.e. `fin` is not `NULL`), then `fin` can be thought of as a "handle" on the file which is passed to an appropriate I/O routine to access the data. In our case, a character is read from the stream using the standard library function, `getc()`:

```
ch = getc(fin)
```

The function, `getc()`, reads a character from the stream accessed by the file pointer, `fin`. It returns the value of character read if successful, and `EOF` otherwise. In the program, each character read is examined to see if it is a digit; if it is, the count for that digit is incremented. Once the end of input file is reached, the file is closed with the statement:

```
fclose(fin);
```

Finally, the program prints the results accumulated in the array.

Let us now examine some details. When a file is opened, it is associated with a file buffer that serves as the interface between the physical file and the program. A program reads or writes a stream of characters from or to a file buffer. A file stream (buffer) pointer must be maintained to mark the next position in the file buffer. This information is stored in the data structure, of type `FILE`, pointed to by the file pointer. Once a physical file is opened, i.e. associated with a file buffer, and a file pointer is initialized, a program uses only the file pointer.

The derived data type, `FILE`, is defined in <`stdio.h`> using a typedef statement, and contains information about a file, such as the location of a file buffer, the current position in the buffer, file mode (read, write, append), whether errors have occurred, and whether an end of file has occurred. Users need not know the details of this data structure, instead, it is used to define pointer variables to a FILE type data item to be accessed by the library functions. For example,

```
FILE *fin, *fout;
```

declares two file pointer variables, `fin` and `fout`. It is now possible to associate these FILE pointers with desired physical files. We use the terms stream and file pointer interchangeably with FILE pointer. Standard files are always open and standard file pointer variables are available to all programs. They are named `stdin`, `stdout`, and `stderr`.

The process of opening a file connects a physical file and associates a mode with the `FILE` pointer. The mode specifies whether a file is opened for input, for output, or for both. The file open function, `fopen()`, associates a physical file with a file buffer or stream and returns a `FILE` pointer that is used to access the file. Here is the prototype:

```
FILE * fopen(char * fname, char * mode);
```

The mode string, `"r"`, specifies that the file is to be opened for reading (i.e. an input file), `"w"` specifies writing mode (i.e. an output file), and `"a"` specifies append mode (i.e. both an input and an output file). If the file was opened successfully, `fopen()` returns a pointer that will access the file stream. If it was not possible to open the file for some reason, `fopen()` returns a `NULL` pointer (a pointer whose value is zero — in C, the zero address is guaranteed to be an invalid address). It is the programmer's responsibility to check to see if the returned pointer is `NULL`. The most common reason why a file cannot be opened for reading is that it does not exist, i.e. an erroneous file name has been used.

Once a file is opened, the library function, `getc()`, reads single characters from the file stream. The argument passed to `getc()` must be a file pointer, and it returns the (integer) value of a character read or `EOF` if an end of file is reached.

Files should be closed after their use is completed. Failure to close open files may destroy files if a program terminates prematurely. The library function that closes a file is `fclose()`, whose argument must be a `FILE` pointer. The process of closing a file frees the file buffer.

In the above program, we specified the name of the input file in the code itself. If the program is to be used with any other input file, we would have to modify the program and recompile. Instead, a flexible program should ask the user to enter file names as needed.

Our next task is to copy one file to another. The algorithm is: simple.

```
get input and output file names
open files for input and output
while NOT EOF, read a character ch from input stream
     write ch to output stream
close files
```

The library routine, `putc(ch, output)` writes a character, `ch`, to a file stream, `output`. The program is shown in Figure 8.7.

Sample Session:

```
***File Copy Program - Character I/O***

Input file :  ccopy.c
Output file :  xyz.c
File copy completed
```

```
/*  File: ccopy.c
    This program copies an input file to an output file one
    character at a time. Standard files are not allowed.
*/
#include <stdio.h>

main()
{   FILE *input, *output;
    char infile[15], outfile[15];
    signed char ch;

    printf("***File Copy Program - Character I/O***\n\n");
    printf("Input file : ");
    scanf("%s", infile);
    printf("Output file : ");
    scanf("%s", outfile);

    input = fopen(infile, "r");
    if (input == NULL) {
         printf("*** Can't open input file ***\n");
         exit(0);
    }

    output = fopen(outfile, "w");
    if (output == NULL) {
         printf("*** Can't open output file ***\n");
         exit(0);
    }

    while ((ch = getc(input)) != EOF)
         putc(ch, output);
    fclose(input);
    fclose(output);
    printf("File copy completed\n");
}
```

Figure 8.7: Code to copy one file to another

The program first reads the input and output file names. We use `scanf()` to read the file names into strings, `infile` and `outfile`. These files are then opened for input and output, respectively. If either of the files cannot be opened, an error message is printed and the program is terminated by an `exit()` call. If both files are opened successfully, the copying is done in a loop until `EOF`. The loop reads a character from input into `ch` which is then written to the stream indicated by `outfile` using `putc()`. When `EOF` is reached, the files are closed and a message printed.

The file routines, `getc()` and `putc()` can be used with standard files as well. We just use the predefined file pointers for the standard files:

```
ch = getc(stdin);
putc(ch, stdout);
```

The above programs terminate if an attempt to open a file is unsuccessful. As an improvement to these programs, friendly programs should allow the user to rectify possible errors in entering file names.

8.3.2 Formatted I/O

When we read or write numeric data from or to standard file streams, `scanf()` and `printf()` convert character input to internal numeric values and vice versa. Similar functions are available for non-standard files. The function, `fscanf()` reads formatted input from a file and `fprintf()` writes formatted output to a file. The only difference between `scanf()`, `printf()` and `fscanf()`, `fprintf()` is that the latter require an additional argument which specifies the input file stream. For example, to read and write an integer from and to a file stream, we use:

```
fscanf(inp, "%d", &n);
fprintf(outp, "%d", n);
```

where `inp` and `outp`, are `FILE` pointers. The other arguments are the same as those for `scanf()` and `printf()`; the format string gives the conversion specifications, and the arguments that follow reference the objects where data is to be stored or whose values are to be written. The return value of `fscanf()` is the same as `scanf()`: number of items read or `EOF`.

Our next task is to read exam scores into an array from a file and determine the average, the maximum, and the minimum. It is assumed that the data file of exam scores is prepared using an editor. The algorithm is simple enough:

```
get input file name
open input file
read exam scores into an array
process the array to find average, maximum, and minimum
```

We will use a function, `proc_aray()`, to process the array. It will return the average but will indirectly store the maximum and minimum values in the calling function. The program is shown in Figure 8.8.

The sample session assumes that the scores are in an input file `scores.dat` prepared using an editor and shown below:

```
/*  File: avgfile.c
    This program reads exam scores from a file and processes them to
    find the average, the maximum, and the minimum.  */
#include <stdio.h>
#define MAX 100
float proc_aray(int ex[], int lim, int *pmax, int *pmin);
main()
{   int max, min, n, lim = 0, exam_scores[MAX];
    char infile[15];
    float avg;
    FILE * inp;

    printf("***Exam Scores: Average, Maximum, Minimum***\n\n");
    printf("Input File: ");
    scanf("%s", infile);
    inp = fopen(infile, "r");
    if (!inp) {
         printf("Unable to open input file\n");
         exit(0);
    }
    while (lim < MAX && fscanf(inp, "%d", &n) != EOF)
         exam_scores[lim++] = n;
    fclose(inp);
    if (lim == 0) exit(0);
    avg = proc_aray(exam_scores, lim, &max, &min);
    printf("Average = %f, Maximum = %d, Minimum = %d\n",
              avg, max, min);
}

/*  This function computes the average of an array, the maximum and
    the minimum. Average is returned, the others are indirectly
    stored in the calling function.  */
float proc_aray(int ex[], int lim, int *pmax, int *pmin)
{   int i, max, min;
    float sum = 0.0;

    max = min = ex[0];
    for (i = 0; i < lim; i++) {
         sum += ex[i];
         max = ex[i] > max ? ex[i] : max;
         min = ex[i] < min ? ex[i] : min;
    }
    *pmax = max;
    *pmin = min;
    return sum / lim;
}
```

Figure 8.8: Code for avgfile.c

```
67
75
82
69
```

Sample Session:

```
***Exam Scores:  Average, Maximum, Minimum***

Input File:  scores.dat
Average = 73.250000, Maximum = 82, Minimum = 67
```

The driver opens the input file and reads data into the array, `exam_scores[]`. The number of elements are counted by `lim`. If `lim` is zero, the program is terminated; otherwise, the program calls `proc_aray()` to process the array for the average, the maximum, and the minimum. In the call to `proc_aray()`, `main()` passes as arguments `exam_scores`, `lim`, and pointers to `max` and `min`.

The function, `proc_aray()`, initializes values of local variables, `max` and `min`, to the value of the first element of the array, `ex[0]`. It then traverses the array, maintains a cumulative sum of the scores, and updates the values of `max` and `min` using the following conditional expressions:

```
max = ex[i] > max ? ex[i] : max;
min = ex[i] < min ? ex[i] : min;
```

Here, if an array element, `ex[i]`, is greater than `max`, `max` is assigned `ex[i]`; otherwise, `max` is assigned `max`. Similarly, the minimum is updated when an array element is smaller than the minimum. Finally, the function indirectly stores values of maximum and minimum, and returns the value of the average score.

8.4 Common Errors

1. Use of `scanf()` to read strings with white space. When `scanf()` is used to read a string, only part of an input string may be read: it skips over leading white space, and reads a string until the next white space.

   ```
   scanf("%s", msg);

   Input: this is a string
   ```

   With the above input, `scanf()` will read "this", and NOT the whole string, into memory pointed to by `msg`. However, `printf()` will print the entire string until the terminating `NULL`.

8.5 Summary

In this Chapter we have discussed various features available to the programmer in the C standard library. While we have used some of the functions in previous chapters, particularly those for I/O, we have given a more detailed description of the library, and the standard I/O routines

provided there. We have seen that frequently used operations on characters for classifying or converting which we have written for ourselves in the past, are available from the library. In addition, routines for common math operations are also provided in the math library (which may not be automatically linked by the compiler). We have given a few short programs illustrating the use of some of these functions. A more complete list of available library routines is provided in Appendix C.

We have also given a complete description of the formatted I/O functions, `scanf()` and `printf()` detailing the options available for formatting input and output. Finally, we have discussed variations on the I/O routines available, both for characters and formatted, which allow direct access to data in files from within a program. These new routines include `getc()`, `putc()`, `fscanf()`, and `fprintf()`, as well as functions for managing connection to the physical files: `fopen()` and `fclose()`.

The full power of the C standard library is now available for future program development in later chapters.

8.6 Exercises

1.
```
main()
{    long n;

     scanf("%d", &n);
}
```

2.
```
main()
{    long n = 12L;

     printf("%d\n", n);
}
```

3.
```
main()
{    double x;

     scanf("%f", &x);
}
```

4. If `x` is 100 and `z` is 200, what is the output of the following:

```
if (z = x)
     printf("z = %d, x = %d\n", z, x);
```

8.7 Problems

1. Write a program to make a table of decimal, octal, and hexadecimal unsigned integers from 0 to 255.

2. Write a program to print a calendar for a month, given the number of days in the month and the day of the week for the first day of the month. For example, given that there are 30 days and the first of the month is on Tuesday, the program should print the calendar for the month.

```
      CALENDAR FOR THE MONTH
sun   mon   tue   wed   thu   fri   sat
            1     2     3     4     5
6     7     8     9     10    11    12
13    14    15    16    17    18    19
20    21    22    23    24    25    26
27    28    29    30
```

3. Write a program to read the current date in the order: year, month, and day of month. The program then prints the date in words: Today is the nth day of Month of the year Year. Example:

```
Today is the 24th day of December of the year 2000.
```

4. Write a program that prints a calendar for a year given the day of the week on the first day of the year. (Use Problem 3.6 for the definition of a leap year).

5. Write a program that prints a calendar for any year in this century given the day of the week for the first day of the current year.

6. Write a function that returns the value of a random throw of two separate dice.

7. Write the following functions:

```
first\_card() that draws a random card from a full deck.
second\_card() that draws a random card from the remaining deck.
Similarly, write third\_card() and fourth\_card().
```

   For the last three functions, you will need arguments that indicate what cards have already been drawn from the deck.

8. Write a program using the functions of Problem ? to play a game of "black jack" with the user. Each side is dealt cards alternately. First each side is dealt two cards, but one at a time. Then, if necessary a maximum of one more card is allowed for each player. The player with the highest score, not exceeding 21, wins. In a tie, the user wins. The program should reshuffle the cards and play the game as long as the user wishes. The score is obtained by summing the value of each card. The value of a card is the face value of the card, except that an ace can be either 1 or 11 and all picture cards are 10.

9. Randomly toss a coin: repeat and count the number of heads and tails in 100 tosses, 500 tosses, 1000 tosses.

10. Write a program to play a board game with the user. The game uses a throw of two dice. The rules of the game are as follows. Each player takes a turn and is allowed a succession of throws. If a player's first throw is seven or eleven, he/she loses the turn. Otherwise, the player's score is increased by the value of each throw until the dice show a seven or a eleven. The turns continue between the user and the program until a pre-set limit for the score is reached.

11. Write a program to compare the routine sq_root() written in Chapter ? with the standard library routine. How close are the routines?

12. Write a function that returns the hypotenuse, given the two sides of a right angled triangle. A hypotenuse is the square root of the sum of the squares of the two sides of a right angled triangle.

13. Find all the angles of a right angled triangle if the lengths of the two sides are given. Since it is a right triangle, one angle is pi / 2. Also, the ratio of the two sides in a right triangle gives the tangent of one of the other angles. Therefore, one angle is the arctangent of the ratio of the two sides. Use a library function to get the arctangent of a value. The other angle is easily obtained since the three angles must add up to pi.

14. Use library routines to compare values of sine, cosine, and exponential with those calculated by Chapter 3 problems 3.1 through 3.3.

15. Write a menu-driven program that handles the grades for a class. The program allows the following commands.

```
    Get data: gets id numbers and integer scores for a set of 3 projects from a
file. Assume that the id numbers start at 0 and go up to a maximum of 99.
    Print data: prints the scores.
    Average scores: averages each set of scores.
    Change scores: allows the user to make changes in scores for any project
and for any id number.
```

16. Write a program that reads a text of characters from a file and keeps track of the frequency of usage of each letter, digit, and punctuation.

17. Write a menu-driven program that reads input data from a file. The program reads and stores for each student the ID number, course numbers, credits, and grades. Assume a maximum of 3 courses per student. The program should compute and store the GPR for each student. At the end of input, the program writes to a file as well as to the standard output all the input data and GPR for all students.

18. Write a program that shuffles and deals out all 52 cards of a deck of playing cards to 4 players. Each card is dealt in sequence around a table to players in the following order: west, north, east, south. Print out the hands of each player. You must use a random generator, but discard a possible card if it has already been dealt out. Use an array of 52 elements to keep track of the cards already dealt out.

19. Write a program to play the game of 21 with a limit of five cards for each player. Assume the program plays south and deals the cards. The other three players are in order west, north, and east. Cards must be dealt randomly.

20. Write a program that reads a positive integer n; it then generates a set of n random numbers in a range from 0 to 99. Store and count the frequency of occurrence of each number. Print the frequency of each number.

21. Use an array to read from a file and store the sample values of an experiment at regular intervals. Plot the graph of the sample values versus time. Time should increase vertically downwards. Use '*' to mark a point. Write a program to read in sample values and call a function to plot the values.

22. Repeat Problem 27, but plot a bar chart for the sample values.